Microcomputers in Civil Engineering 8 (1993) 175–198

# An Intelligent Vehicle Highway Information Management System

Shashi Shekhar,[a] Toneluh A. Yang[a] & Peter A. Hancock[b]

[a]Computer Science Department, [b]Human Factors Research Laboratory, University of Minnesota, Minneapolis, Minnesota 55455, USA

**Abstract:** *An IVHS (Intelligent Vehicle Highway System) information management system obtains information from road sensors, city maps and event schedules, and generates information to drivers, traffic controllers and researchers. We extend the relational database to model traffic information in a relational database by abstract data types and triggers. Abstract data types are needed for efficient modeling of spatial and temporal information, which create inefficiencies in traditional databases. We use monotonic continuous functions to map the object locations to disk addresses to save disk space and computation time. Model of spatial data is created to efficiently process moving objects. We provide schema for IVHS databases with the relevant abstract data types. We also have a large sample of the relations needed to model IVHS data. Several interesting queries are presented to show the power of the model. Triggers are defined, using rule-definition mechanisms to represent incident detection and warning systems. An efficient physical model with MoBiLe access method is provided.*

## 1 INTRODUCTION

We are designing a traffic information base for Intelligent Vehicle Highway System (IVHS) application to create a shared resource, efficient disk-based computation and integrity of data. As shown in Fig. 1, the information stored in the IVHS database will be used by transportation system designers for traffic modeling and control. The same information will be used by human-factors researchers to simulate driving conditions in intelligent cars with headsup displays and onboard computers. Efficient disk-based computation is needed to detect collisions and traffic incidents. Spatial access methods, rules and triggers, provided by the database system, will be used for efficient disk computation of incident-detection. One of the important benefits a database may offer is the integrity of stored data. Changes (deletions or additions) of maintenance work on highways, for example, shall not leave any management holes under the database's integrity constraints.

A geographical database is the major component of an IVHS information management system, which obtains information from different sources and sends the processed information to different clients. The sources may include traffic reports, scheduled traffic events, and static information (e.g., maps, incident descriptions, etc.). The processed information may go to transportation system designers, drivers, traffic controllers, psychological experimenters, etc. The data sources include city maps, road maintenance schedules, periodic sensor data from various locations, and traffic reports. The data is characterized by its spatial and temporal nature. Furthermore, sensor data leads to high update rates. The clients of the database include

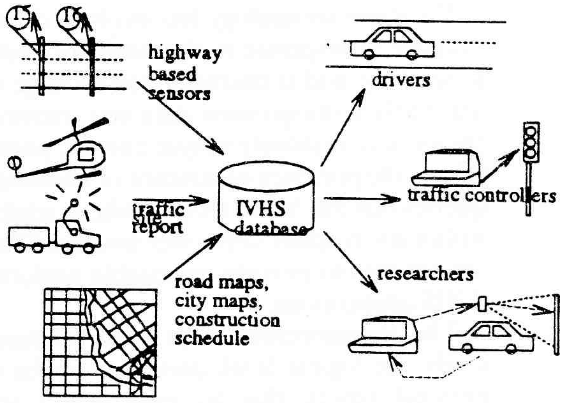

**Fig. 1.** Data sources and clients of an IVHS database.

drivers on the road, traffic controllers and transportation researchers. Drivers may be interested in finding short and uncongested paths to destinations. Traffic controllers may be interested in incident detection and management as well as traffic-flow control. Researchers often wish to validate their models (e.g., traffic-flow models, driver-behavior models) with live data from sensors.

Due to the diversity of traffic information, an IVHS database requires special capabilities for solving problems which the traditional database systems need not consider.

First, geographical data such as highway construction blueprints and city maps comprise a large part of the traffic data. An IVHS database should be capable of representing, processing, and displaying static geographical data. The buildings in the city are static and do not have overlapping area; the sizes of the areas occupied by buildings are comparable and do not vary a great deal. The areas occupied are rectangular in shape, with comparable lengths on the opposite sides. The object population density is high near the city center and declines with the distance from the city center. Some cities have more than one city center, which can be modeled as a linear combination of simple cities with one city center. The database should be able to store geographic objects by preserving their geographical proximity.

Second, traffic is a dynamic entity, and it may be modeled with a large number of timestamp measurements, producing large volumes of data. The macroscopic view of traffic may raise issues related to temporal databases. To simplify and focus our discussion, we will explore the microscopic view of traffic by examining individual vehicles in the traffic. Individual vehicles move and change their locations creating a sequence of updates to the database to update their locations for future spatial queries. An IVHS database must provide access methods to handle these updates due to motion.

Database technology has evolved over the last two decades in response to the needs of commercial data processing, and is characterized by large record-oriented, fairly homogeneous data sets, mostly retrieved in response to relatively simple queries: point queries that ask for the presence or absence of a record, and interval queries that ask for all records whose attribute values lie within given upper and lower bounds. These databases are not able to provide reasonable performance for the IVHS applications.

The transportation data must be modeled at two levels: the logical level, presented to the user, and the physical aspects that an implementer sees. Consider the example where a spatial object (e.g. a road intersection) is stored in a database by using the boundary-representation approach. In this approach, a two-dimensional region is represented as a collection of edges, and edges are represented by their endpoints. Three relations (regions, edges and points) may be used to model the data. A relational database may be designed with the following relations:

*region:* a pair $(R_k, e_i)$ identifies a region $R_k$ and one of its edges $e_i$.
*edges:* a triple $(e_i, p_m, p_n)$ identifies an edge $e_i$ and its two end points $p_m$ and $p_n$.
*points:* a triple $(p_n, x, y)$ identifies a point and its coordinates $x, y$.

The representation smashes as simple an object as a square into parts, spread over different relations, and therefore over the storage medium. The question whether a region intersects a given line $L$ is answered by intersecting each of the edges of the region with the line $L$. If the pair $(R_k, e_i)$ in the relation *region* contains the equation of the edge, the intersection between edge $e_i$ and $L$ can be computed without accessing other relations, but to determine whether the intersection point lies outside or inside the edge $e_i$ requires accessing the relations *edge* and *point*, i.e., accessing different storage blocks, resulting in many more disk accesses than the problem requires.

Efficiency requires, at least, retrieving all the data that defines a basic region such as square as a unit. In addition, geometric properties may be used to answer proximity queries more efficiently. A sample region, such as a rectangle, can be represented by 8 parameters that define the coordinates of four corner point. Alternatively, rectangles may be specified by an enclosing circle with common center, with the angles subtended by each corner. The second approach takes 7 parameters (2 coordinates for the center of circle, five for the radius and corner points). The negative answer to many intersection queries can be computed very efficiently by comparing the radius with the distance between the line and the center of circle. In essence, if geometric objects are merely considered logical entities, we fail to take advantage of the rich structure of geometry. The physical design of a database must utilize geometric properties for computation efficiency.

In recognition of the fact that modeling is insufficient for spatial databases without efficient physical storage, intense efforts to design data structures for spatial data have been made in recent years. A number of spatial-data access methods have been proposed to retrieve objects which are $n$-dimensional points or solids. The spatial access methods optimize queries to retrieve all points or solids enclosed in or overlapping with a given search region. The proposed access methods include R-tree,[7,23] Grid Files,[1,5,8,17,18] and other search trees.[4,10,11,14,15,19] However, these access methods were designed with the assumption of static worlds with no moving objects. Update rates were assumed to be much smaller than search query rates. The object shape, size and location distributions are assumed to be random rectangles with a high degree of overlap, characteristic of VLSI/CAD data. Due to random object population, the methods rely on tree-structured directories/indexes, which are balanced after each update to keep the cost of search queries small.

In presence of high update rates due to moving objects, traditional methods incur very high overhead of maintaining the tree-structured indexes. Furthermore they are not able to use the population distribution information to simplify the access methods. Grid-based schemes need large memory buffers to contain their directories and may not preserve the geographic proximity relationship. Hashing-based schemes do not suffer the problem with memory buffers. They, however, do not preserve proximity relationships among geographic objects.

We proposed a new access method called MoBiLe Files (MOnotonic Bounded mapping in LEotard Files) to provide an efficient access method for IVHS applications.[24] The MoBiLe Files method uses the population distribution of the static geographical data in an IVHS domain as the mapping functions between the geographical domain and the storage space. The usage of population distribution information enables efficient data access and storage. It does not have the problem of directory structure maintenance, and most importantly, preserves the proximity relationship among geographical objects.

The rest of the paper is organized in the following manner. The requirements of an IVHS application are discussed in Section 2. We present our IVHS database schema in Section 3. Entities in an IVHS application are presented first, and abstract data types are then defined to model these entities. In Section 4, query languages of the IVHS database are described and sample queries are presented. We discuss the mechanism of integrating knowledge into an IVHS database in Section 5. Access methods are discussed in detail in Section 6. We describe the related literature and identify our contributions. We describe MoBiLe functions and compare the proposed access method with major families of traditional access methods by detailed cost modeling and experimentation. The experimental parameters, configurations, and the results are described. Finally, we present our conclusions and recommendations for future work.

## 2 REQUIREMENTS OF IVHS DATABASE

Five key characteristics distinguish IVHS data management from the other applications:

1. Data from individual sensors represent a stream of values, ordered by time of sensing. The values are assessed by their ordering in time. Data values associated with current and recent time are used more often than are older data values. The average value over time interval is often computed.
2. Many objects (e.g., sensor, buildings, roads) are embedded in $k$-dimensional Euclidean space. For example, the record may be considered to be a point in attribute space. However, attribute space is not Euclidean space, since the distance between two points (e.g., two names) may neither be meaningful nor satisfy the triangle inequality.
3. The objects are often accessed through their location in space. For example, partial match and orthogonal range queries are common in traditional applications. In contrast, queries for overlapping between two regions are more popular in IVHS databases.
4. The shape of a typical spatial object may be fairly complex. Although a record in a traditional database may contain a lot of attributes, for searching purposes, it resembles a point in the attribute space. A typical spatial object, on the other hand, may be a region of complex shape, and we may need to reduce them to predefine primitives such as points, edges, or triangles.
5. An IVHS data processing system needs to store and process information traditionally represented via maps.[16] Sample data include the Landsat image data bank,[30] census data,[25] and a city map of roads and buildings. Another important set of data includes a schedule of important events, such as a road maintenance schedule, and an event schedule for major traffic sources including stadiums, shopping centers, etc.

An important problem in designing IVHS data processing systems is efficient physical data modeling. Traditional data modeling techniques do not provide an adequate means for dealing with IVHS information. Most IVHS data must be qualified by the location where it is valid, the time of observation and its accuracy.[20] It is generally believed that current general purpose databases are inadequate to deal with spatial information mostly due to efficiency problems, and partly due to lack of direct support for spatial and temporal concepts.

Two semantic domains that are essential to dealing with IVHS concepts, space and time, are provided by the formalism. Without excluding alternate views of space and time, we provide a kernel set of spatial and temporal concepts and operators. Based on spatial and temporal logic, they allow one to state that a fact is true at some point in time and in a particular place.

The modeling of spatial objects is divided into the following parts: representation of space, representation of spatial objects, embedding objects in space, object transformations (translation, rotation), preservation of Euclidean proximity relationships, and aggregation operations.

The representation of space via a coordinate system allows naming and reference to interesting parts of space. For example, to retrieve roads under maintenance in downtown Minneapolis, we need to identify the span of the area named 'downtown Minneapolis'. The representation of objects allows us to model the shape of interesting objects such as roads, buildings, and vehicles. An embedding of objects in the space provides information about the space currently occupied by the object. For example, to answer queries relating to collisions between two vehicles, we need to examine the overlap of space occupied by the vehicles. The separation of object representations from their embedding in space is helpful in simplifying the computation of embedding-independent object properties. For example, the volume of a vehicle can be computed without the knowledge of its location. Using such transformations as translation and rotation, we can represent the motion of a vehicle and compute new embeddings.

Preservation of spatial proximity between object pairs is needed to efficienctly answer proximity queries like collision between vehicles. It reduces the computer disk accesses needed to answer the queries by simplifying the representation of the object needed to answer most queries. Other frequent queries, such as finding a path from the airport to a downtown hotel, a boundary traversal to locate all objects close to downtown Minneapolis, or sweep algorithms to scan all objects along a road, also dictate the physical model of spatial objects.

Our formalization views the world as a collection of entity instances. Information about individual objects is captured via the set of attributes defining its properties. For example, the EE/CSci building is characterized by its location, office hours, capacity and other traffic-relevant attributes. Related individuals are grouped into entities represented as a table with one column for each attribute and one row for each individual. All buildings are grouped in one table. All roads may be grouped in another table.

We classify the entities into two classes: materialized and virtual. Materialized entities are stored in the database. Virtual entities are not stored, but can be computed from the stored entities. For example, an intersection is a virtual entity. Given the identification of intersection in terms of the roads which meet there, interesting properties (e.g., location) of individual intersections can be computed.

Spatial and temporal aspects of data are modeled via a set of data types which specify the spatial and temporal attributes and operations. The space is modeled via a rectangular coordinate system. Objects are modeled approximately by a collection of primitive objects, which include rectangles and rectangular solids. The embedding of objects in space is modeled by the coordinates of the center of the object. Translation and rotation operations are supported and modeled by altering the values of relevant attributes of the objects, representing the new embedding. Proximity relationship is preserved via the MoBiLe mapping function, which determines the disk address of an object from its spatial coordinates. The mapping function is monotonic and continuous to preserve proximity relationships. The boundary traversal and other algorithms are supported efficiently by the mapping.

Consistency among the stored data is specified via a set of integrity constraints that identify the constraints among values for attributes of individual objects. For example, the office hours for EE/CSci buildings may be 8 to 10 h per day (not typical of many Computer Science Departments!). The areas occupied by two independent buildings are non-overlapping.

An important constraint in defining the formalism has been the desire to implement it in an extensible database such as Postgres. Postgres provides a template data base which can accept user-defined data types and operators to model IVHS applications. We are implementing the formalism on Postgres version 3.0 in a Unix environment on Sun Sparc machines using C and Lisp. Graphic interface for the map data is provided from the Xfig and Pic tools. The purpose of the experiment is to identify ways to overcome the limitations of traditional database systems in areas of efficiency, modeling and user interface.

## 3 DATABASE SCHEMA: REPRESENTING THE ENTITIES IN THE TRAFFIC WORLD

A database schema models application domain as a collection of entities with attributes and the relationships among the entities. To represent a domain efficiently, several data models have been proposed. Among them are the network model, the hierarchical model, the relational model, the entity-relationship model, the functional model, the semantic model and the object-oriented model. In addition, variants and extensions of the above models also exist. The extensible relational model,[2-26] for example, has been a main research area since the emergence of the relational model.

We chose an extensible relational database management system, Postgres,[27,28] to represent the schema of the IVHS database. It provides an application database designer with the ability to model both the semantic and the procedural aspects of an application domain.

The choice of model was dictated by the following considerations:

1. The query language should provide commands for defining new types and operations. Database designers can define domain-specific abstract data types with the commands to model the application domain entities.
2. The model should support rules. Rules makes the creation of triggers/demons possible, which are central to processing such events as collision detection and accident monitoring.
3. The model should allow inclusion of new spatial and temporal indexing methods for efficiency and for tailoring to the application-specific computations. The IVHS database domain, for example, contains geographical objects. Since geographical objects usually contain multi-dimensional attributes (e.g., the $X$ and $Y$ coordinates representing locations, or the rectangle representing an area), the traditional one-dimensional accessing schemes widely used in commercial applications are not appropriate for processing geographical data. It is important for the performance of a IVHS database to have specific access methods for representing and accessing geographical objects. We have created a powerful spatial access method called MoBiLe Files, which are capable of mapping from objects' geographical locations to the appropriate physical disk location, based on the population density of the domain. As will be shown in Section 7, MoBiLe Files perform well in terms of both disk space and computing-time usage. With the *define index* command provided by POSTQUEL, we are able to build the MoBiLe File access method into the IVHS database.

The first step of creating an extensible database application is to model the entities in the application domain as abstract data types (ADTs). An ADT contains attributes and operators specific to the type. A circle, for example, contains a center and a radius. A circle can be modeled as an ADT containing two attributes: a center coordinate (centerX, centerY) and a radius. In addition, operations such as 'get_circle_center (circle)', 'get_circle_radius (circle)', 'circle_equal (circle1, circle2)' may be defined for the efficient manipulation of circles. An operation is usually defined as a function taking one or more parameters. The 'circle_equal' operation, for example, is a function taking two circles and returning a boolean value (equal/unequal or True/False).

The entities existing in a IVHS database such as buildings, vehicles, etc. can be modeled as ADTs. In the database model, ADTs correspond to *types* and *relations*. The commands *define type, define C function, define function, define operator,* and *create* were used for the creation of abstract data types. The entity *vehicle*, for example, was modeled in the database as containing five attributes: ID, name, type, box, and safety_envelope (see the *vehicle* relation table in the Appendix for examples). An entity is defined in such a way that the defined attributes and operations are useful for modeling the application domain. One of the purposes of IVHS databases is to detect collision, thus a vehicle contains the attributes 'box' and 'safety_envelope'. As another example, the entity 'building' contains five attributes: building ID, building name, business, box, and schedule. Users may query the schedule of a particular building or jointly query how many buildings have business hours falling within a particular time segment (e.g., 1200–1600 h on 24 Dec).

## 3.1 Entities

We model the sample IVHS data with the following entities: vehicle, building, traffic_sign, traffic_area, traffic_location, sensor, road, bridge, congestion, collision, and event. Each of the entities is shown as a table in the Appendix. Each table contains several columns representing attributes of the entity. Each column has a type, which is offered when building an entity into the database. The entity *vehicle*, for example, could be built into the database using the following command.

```
create vehicle (ID = integer, name = string, type
    = TYPE,box = string, safety_envelope = string)
```

The type of each attribute appears after the '=' sign. A type specifies the domain of a column. Built-in types such as integer, character string are directly specified. The application-defined types such as *box, lines, points, absolute time*, etc., which are described in the next section, are represented using the keyword TYPE. The 'type' column of a vehicle relation, for example, has type *box*. This means a vehicle is represented as a rectangle containing four corner points. This kind of

MER (minimum enclosing rectangle) representation is appropriate for collision detection by checking box overlapping. The function *box_overlap* is an operation specific for the ADT *box*. It takes two boxes as parameters and returns a boolean value. To check if a vehicle's enclosing rectangle overlaps with a building's enclosing rectangle, we could issue the following function call inside a query: box_overlap(vehicle.box, building.box).

## 3.2 Data types and operations

The semantic domain of a traffic information system can be classified into two classes: the spatial (geographical) domain and the temporal domain. The spatial objects are modeled with entities such as points, line segments, paths and boxes. The space is modeled by a coordinate system to embed objects in space. The temporal domain is modeled by time point (absolute time), time interval, periodic time (e.g., every Wednesday, every day at 0900 h, etc.), and schedule. The spatial and temporal classes are modeled as types and are used together with the primitive types to specify the type of domain of columns in a relation.

Spatial data types include *point, box, lseg (line segment),* and *path. Box,* for example, is used in the relations *vehicle, building,* and *traffic_area* to specify the extent of the entities. The box attribute specifies the extent of area covered by a vehicle or building. The box attribute is useful for collision detection. As another example, the type *lseg* and *path* is used to represent *road* and *bridge,* which is either represented as a line segment or as a combination of line segments (path).

Temporal data types contain *abstime* (absolute time), *reltime* (relative time), *tinterval* (time interval), *wctime* (wild card time), *winterval* (wild card interval), and *schedule. Wcinterval,* for example, is used to define the relation *construction,* while *schedule* is used to model the business hours of a building in the *building* relation. Rush hours can also be modeled using the *winterval* data type.

Data types and their definitions are shown in Table 1.

While 'absolute time/interval' is self-explanatory, 'wide card time/interval' needs some explanation. Frequently, in a real-world application, a user needs to specify a generic time or interval without specifying the date or month. The business hours of a building, for example, are usually represented as generic time intervals (e.g., Mon.-Fri. 0800 h to 1600 h, Sat. and Sun. 1200 to 1500 h). Wild card time/intervals (*wctime, winterval*) serve the need of specifying a generic time/interval or schedule. It is defined using the wild card symbol ?, which matches all values for any fields (day, month, hour:minute:second, year) in a time expression. A time expression has the following format: 'month day hour:minute:second year'. If a user specifies a time expression using the ? symbol as follows ["?? 16:00:00 1991", "?? 20:00:00 1991"], the user tells the database to look at every winterval 1600 h to 2000 within the year 1991, regardless of the dates. The following sample query specifying a generic time interval.

query # 1: All buildings near a camera location which generated traffic during 1600–1800 h in 1991.

```
retrieve (building.name)
from b in building, c in sensor
where c.class = "camera"
and near(b.box, c.point, 3)/*e = 3 miles */
and in(end(b.offhrs), ["??16:00:00 1991", "??
18:00:00 1991"]).
```

The *Define type* command is used to build types into the database. The spatial and temporal data types are

**Table 1**
Data types

| | *Name* | *Definition* |
|---|---|---|
| *Spatial types* | | |
| | point | Data point with two coordinates $(x, y)$ |
| | box | Two dimensional rectangle (represented by four corner point) |
| | lseg | Line segment (represented by two end points) |
| | path | Curve approximation by a variable length array of line segments |
| *Temporal types* | | |
| | abstime | Time instant (year, month, date, and day-time) |
| | reltime | Relative time instant with respect to now |
| | tinterval | Time interval (starting abstime, ending abstime) |
| | wctime | Absolute time specification with wild card and defaults |
| | winterval | Time interval (starting wctime, ending wctime) |
| | schedule | A list of wintervals |

depicted in Fig. 2 along with a sample relation, *building*. The upward arrow points to the type specifier for the particular column in the relation. The columns without upward arrows have primitive types such as integer or character strings.

Each data type has a set of operations defined for easy and efficient use of the data type. The *box* data type, for example, has the following operations: *box_overlap()*, *inside()*, *box_center()*, *passesVia()*, *near()*, *enclosure()*, and *adjacent()*. The same operation may take different parameters. The operation *near()*, for example, has four different parameter pairs: point/box, box/box, lseg/box, and path/box. Although having the same name, each individual operation is actually defined differently; however the user does not need to worry about the details. This is one of the strong features of abstract data types.

Operations for spatial types are listed in Table 2, and temporal operation are in Table 3. Operations are defined for each data type using either *define C func-*

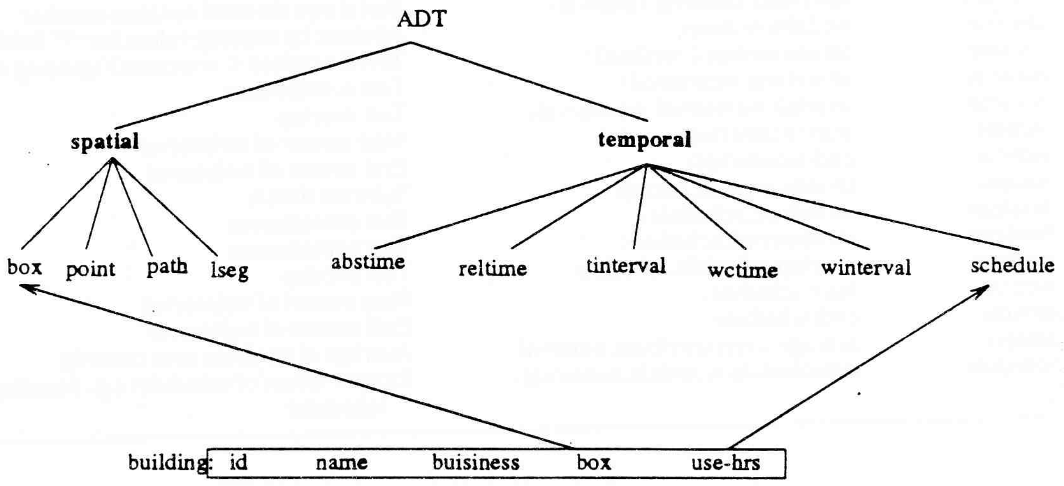

**Fig. 2.** Traffic domain data types.

**Table 2**
Spatial operations

| *Return type* | *Name* | *Meaning* |
|---|---|---|
| boolean | box_overlap(box, box) | Test for overlapping boxes |
| boolean | inside(point, box) | Test if point is in the box |
| boolean | on_ppath(point, path) | Test if point lies on path |
| box | box_center(box) | Return center point of box |
| integer | pointdist(point, point) | Distance between 2 points |
| boolean | inside(box, box) | Test if a box is inside the other box |
| boolean | passesVia(path, box) | Test if a path intersects box boundaries |
| boolean | near(point, point, e) | Test if pointdist(point, point) < *e* |
| boolean | near(point, box, e) | If distance between two arguments < *e* |
| boolean | near(box, box, e) | If distance between two arguments < *e* |
| boolean | near(lseg, box, e) | If distance between two arguments < *e* |
| boolean | near(path, box, e) | If distance between two arguments < *e* |
| boolean | intersect(lseg, lseg) | Test if two line segments intersect |
| boolean | intersect(path, lseg) | Test if path intersects lseg |
| boolean | intersect(path, path) | Test if path intersects path |
| point | intersection(lseg, lseg) | Intersection point |
| point | intersection(path, lseg) | Intersection point |
| point | intersection(path, path) | Intersection point |
| box | enclosure(box, box) | Smallest box containing two boxes |
| box | enclosure(path) | Smallest box containing the path |
| box | enclosure(point, box) | Smallest box containing box and point |
| lseg | lineseg(point, point) | Straight line joining two points |
| boolean | adjacent(path, box) | Test if path adjacent to box |

**Table 3**
Temporal operations

| *Return type* | *Name* | *Meaning* |
|---|---|---|
| abstime | intervalstart(tinterval) | Start instant of tinterval |
| abstime | intervalend(tinterval) | End instant of tinterval |
| abstime | timemi(abstime, reltime) | Subtract timeA |
| abstime | timenow( ) | Returns to current time |
| boolean | abstimele(abstime1, abstime2) | Test if abstime1 < = abstime2 |
| boolean | reltimele(reltime1, reltime2) | Test if reltime1 < = reltime2 |
| boolean | ininterval(abstime, tinterval) | Test containment |
| boolean | intervalov(tinterval, tinterval) | Test overlap |
| boolean | intervalct(tinterval, tinterval) | Test if one tinterval contains another |
| abstime | wc2abs(wctime) | Abstime by copying values for "?" fields from timenow ( ) |
| boolean | timele(wctime1, wctime2) | Test if wctime1 < = wctime2 ignoring wildcard valued fields |
| boolean | in(wctime, wcinterval) | Test containment |
| boolean | overlap(wcinterval, wcinterval) | Test overlap |
| wctime | start(wcinterval) | Start instant of wcinterval |
| wctime | end(wcinterval) | End instant of wcinterval |
| wctime | timemi(wctime, reltime) | Subtract timeA |
| boolean | in(wctime, schedule) | Test containment |
| boolean | in(winterval, schedule) | Test containment |
| boolean | overlap(schedule, schedule) | Test overlap |
| wctime | start(schedule) | Start instant of wcinterval |
| wctime | end(schedule) | End instant of wcinterval |
| integer | average_overt(attribute, tinterval) | Average of attribute over tinterval |
| schedule | subschedule(schedule, winterval) | Extract subset of schedule (e.g., Monday schedule, or April schedule) |

*tion* or *define POSTGRES function* commands. Operations provide access to data types. Type-specific operations are required for efficient and easy data manipulation.

## 4 QUERY LANGUAGE OF THE IVHS DATABASE

Once the application database is created, users can use the types and functions together with the data access commands to retrieve or update information stored in the database.

The query langauge includes commands defining new types and functions not available in traditional database language like SQL or QUEL. These commands include ***define type, define function, define rule, define index***, and ***define operator***. Once a function is defined, it can be used in the queries. The database run-time system will automatically load the corresponding function code when it processes the queries. A function can also be bound to an operator so as to improve the syntax of a query. The *inside(box,box)* function, for example, can be bound to the operator '< =' using the *define operator* command. The user of the database can then issue a more natural expression such as 'box1 < = box2' to replace 'inside(box1, box2)'.

An IVHS database designer should have the ability to add new access methods into the database. The commands *define index* and *define operator* can be used together to build an access method into the database which may not have been provided. As mentioned before, the geographical objects contained in a IVHS database require spatial access methods for efficient processing. Making use of a user-defined index feature, the MoBiLe Files access method can be integrated with the IVHS database. Once an access method has been built into the database, operators will automatically invoke the corresponding access methods, but this complexity is invisible to the user of the database.

Higher-level constructs such as VIEWS and transitive closure are frequently required in a database application. Finding the shortest path between two locations, for example, requires the capability of expressing transitive closure of the query language. We have integrated the transitive closure specification into the view definition command. The command 'define VIEW <view name> (<attributes>) AS TRANSITIVE_CLOSURE of<relation name>BY <view body>' defines a view recursively taking a relation and the view itself building a view containing the transitive closure of particular attributes.

```
define VIEW simplepath (from_building, to_building, road.name, distance)
retrieve (b1.name, b2.name, r.name)
b1 from building, b2 from building, r from road
WHERE
adjacent(b1.box, r.lseg, 20yards),
AND adjacent(b2.box, r.lseg, 20yards)
AND distance = pointdist(box_center(b1.box),
box_center(b2.box)).
define VIEW route (from_building,        to_building,
path, length)
AS TRANSITIVE_CLOSURE OF simple_path
BY
retrieve (s.from_building, r.to_building,
s.distance + r.distance)
from s in simple_path, r in route
where s.to_building = r.from_building.
```

For simplicity, we assume most important sections of roads have important buildings near them. We also assume that the length of road joining two buildings is approximately equal to the distance between the center points of the buildings. With intersections in the database, one could use *pointdist(intersection1.point, intersection2.point)* to improve this estimate.

A general case can be handled by storing important intersections in the relation *traffic_location*, and including the paths between intersections in the VIEW simple_path. This may be necessary for freeway intersections.

query # 2: Find the shortest path from EE/CSci to Min/Dot.

```
retrieve (r.path, min(distance))
from r in route
where r.from_building = "EE/CSci"
and r.to_building = "Min/Dot".
```

The sample query demonstrates use of the 'route' VIEW to find the shortest path between the EE/CSci building of the University of Minnesota and the Min/Dot building. It should be noted that queries containing transitive closure are usually expensive and require query optimization for efficient processing; query optimization has been a heated research area in database systems. An approach to solving transitive closure queries is presented in Ref. 1.

## 4.1 Example queries

First, queries may be classified as relevant to drivers, experiments, or traffic-controllers. The drivers, for example, may want to know whether there exists anything within its safety envelope. The following is an example of a driver query.

query # 3: Find all the objects existing within the safety envelope of the self vehicle.

```
define POSTQUEL function safety_check (vehicle)
returns string is
   retrieve (building.name)
      where box_overlap(self.safety_envelope,
      building.box)
   retrieve (vehicle.name)
      where box_overlap(self.safety_envelope, vehicle.box)
   retrieve (trafficsign.name)
      where inside(traffic_sign.point,       self.safety-
      envelope)
   retrive (cd_list = safety_check(self))
```

As another example, the driver may want to know all the gas stations within one mile of the road on which he is driving. The following query is an example of this type of query.

query # 4: Find all gas stations within 1 mile of the road adjoining Minneapolis, MN to Rochester, MN?

```
retrieve (building.name, building.location)
   from r in road,
   l1 in location
   l2 in location,
where building.business = "gasstation"
   and l1.name = "Rochester"
   and l2.name = "Minneapolis"
   and passesVia(r.path, l1.box)
   and passesVia(r.path, l2.box)
   and near(building.box, r.path, 100 yards).
```

The traffic controllers may want to query the database for all buildings near a camera location which generated traffic during 1600–1800 h on 1 Jan. 1991.

query # 5: Find adjacent camera pairs on common lane with high difference (50% = 0·5) in average lane occupancy for last 5 min.

```
retrieve (c1.id, c2.id)
from c1, c2 in sensor
where c1.class = "camera"
and c2.class = "camera"
and c1 != c2
and floatmi(
   average_overt(c1.occupancy,        [timenow(),
   timemi(timenow(), @5 minute)]),
   average_overt(c2.occupancy,        [timenow(),
   timemi(timenow(), @5 minute)])
)> = 0·5.
```

The experimenters may want to know if the self vehicle in the driving simulation environment has collided with any objects.

query # 6: Find all the objects colliding with the self vehicle.

```
define POSTQUEL function Collision_De-
tection (vehicle) returns string is
    retrieve (building.name) where box_over-
    lap(self.box, building)
    retrieve (vehicle.name) where box_over-
    lap(self.box, vehicle.box)
    retrieve (traffic_sign.name) where
    inside(traffic_sign.point, self.box)
retrieve (cd_list = Collision_Detection (self))
```

## 4.2 Point, range, aggregate, join, closure and boundary queries

Queries can also be classified using a different set of criteria based on the data types involved. A query may be an instance of point queries, range queries,[12] aggregate queries, join queries, transitive closure or boundary queries.

Most of the above queries can be implemented efficiently. Point and range queries are implemented with the help of spatial and temporal access methods. Aggregate, join, and transitive closure queries, however, are expensive in terms of implementation cost and space and time cost. A transitive closure query, in particular, is very expensive. We are investigating the method of formulating path planning as a function. MoBiLe Files may be used in a rule-based route computation method to reduce the search space of transitive closure queries.

An example of a point query is presented below, where camera locations (represented as point types) are requested.

query # 7: Find a camera location with a high average lane occupancy for the last 10 min.

```
retrieve (c.location)
from c in sensor
where c.class = "camera"
and    average_overt(c.occupancy,   [timenow( ),
timemi(timenow( ), @ 10 minute)]) > 50
```

The following two examples demonstrate range queries.

query # 8: Which roads are under construction near downtown?

```
retrieve (r.name)
from r in road, c in construction, area in traffic_area
where area.name = "Mpls Dntn"
and passesVia(r.path, area.box)
and r.id = c.entity
and ininterval(timenow( ), c.interval).
```

query # 9: List buildings with their locations, in East bank MPLS campus UMN.

```
retrieve (b.name, b.location)
from b building, area in trafficarea
where area.name = "MPLS campus UMN"
and inside(b.box, area.box).
```

A join query asks for the intersection of two set of attributes from two different relations. The following query is an example of a join query.

query # 10: All buildings near a camera location which generated traffic during 1600–1800 h on 1 Jan. 1991.

```
retrieve (building.name)
from b in building, c in sensor
where c.class = "camera"
and near(b.box, c.point, 3)/*e = 3 miles */
and ininterval(interalend(b.offhrs),
    [Jan 1 16:00:00 1991, Jan 1 18:00:00 1991]).
```

Transitive closure queries involve recursion. Finding the shortest path between two locations, for example, requires the database to recursively build a *route* view and select the one with minimum path length. The view definition mechanism and a sample query involving transitive closure were shown as query # 2. The *min()* function in the query is an aggregate function which takes a set of paths and returns the one with the shortest path length.

Boundary queries involving traversing the boundary of a given object. For example, to retrieve all the roads near the University of Minnesota campus, we need to traverse the boundary of the campus to collect the roads. A box is represented in a way that allows us to compute the boundary, as well as the area, efficiently. The following is an example of boundary queries.

query # 11: Find all roads that intersect with the roads (e.g., Washington, University, 4th street, 19thAvS/10th AvSE) adjacent to University of Minnesota, Minneapolis.

```
retrieve (r1.name) from r1 in road, r2 in road,
area in traffic_area
where area.name = "UofM"
and near(r2.path, area.box, 10meter)
and intersect(r1.path, r2.path).
```

## 5 EVENT DETECTION AND DATABASE TRIGGERS

In a IVHS database domain, events such as traffic accidents, traffic congestion and safety envelope violations need to be persistently monitored. Since events do not occur predictably, it is more appropriate to model them as triggers rather than as queries. Triggers are more like persistent queries or demons existing in the database. Triggers are usually implemented as rules, which consist of a 'condition' part and an 'action' part. Once built into the database, rules function like demons and constantly monitor the database state to match the 'condition' part of the rule. If the state satisfies the condition, the 'action' part is fired to conduct pre-defined actions (e.g., report collision). It is apparent that triggers are useful in IVHS database applications, where incident detection and warning systems for drivers are important features.

To integrate knowledge into a database system, two approaches have been proposed: the loose-coupling approach and the tight-coupling approach.[29] Both approaches store the knowledge as rules in a knowledge base. The rules can then be shared, updated, and automatically applied. The former approach uses a knowledge manager separate from the DBMS (data base management system) to manage the rules; the second approach integrates the knowledge manager into the DBMS.

In the first approach, a separate knowledge manager is usually built upon an expert system shell such as OPS5, S1, KEE or Prolog. The rule manager and the DBMS are loosely coupled. One disadvantage of loose-coupling approach is that whenever the knowledge manager requires data from the database, it needs to send a query to the DBMS to obtain the data. In addition, the database user needs to learn the rule-specifying language, which is usually quite different from database languages such as SQL or QUEL.

The tight-coupling approach, on the other hand, integrates rules into the database management system. The integration of rules into a DBMS allows the system to deal with dynamic environment, where data are updated frequently. Another advantage of tight-coupling is that it can be used with a database application which is not partitionable. One of the limitations of the loose-coupling approach is that it needs a partitionable database to make efficient inferences. Since most of the application domains are not partitionable, the tight-coupling approach is more realistic.

Several example rules are shown in Table 4. The *define rule* command is easy to use to build rules into the database. Rule 1 (if collision with the self occurs, then report it and store all the colliding objects), for example, can be defined as follows.

```
define rule collision_report is
on Collision_Detection (self.box)
do report_collision()
```

## 6 ACCESS METHODS FOR GEOGRAPHIC SEARCH AND MOTION

Access methods are data structures used to organize the data file on disk for efficient query processing. The access methods help in retrieving the data records of files in various sorted order to help answer interval queries efficiently.

### 6.1 Relation literature and our contribution

The last decade has seen a surge of interest in the design of spatial access methods for VLSI-CAD and geographic data. A number of access methods have been proposed to handle spatial data and spatial queries. Recently a set of comparison studies have been reported to evaluate alternative access methods for various applications.[6, 13]

Spatial access methods can be classified into two major groups: point access methods and spatial interval access methods,[13] based on the types of queries supported. Point access methods are useful in answering queries about the properties of a point. For example, the elevation at a given geographic location may be modeled as a point query. The interval or spatial query relates to a region of space. An example of region queries would be to determine the number of objects overlapping a given area (e.g., collision detection). We will focus on the access methods for spatial interval queries.

Spatial access methods are based on two canonical ideas: Grid-files and R-tree. Grid-files divide the geographical region into a rectangular grid of possibly

**Table 4**
Examples of application defined rules

| | |
|---|---|
| Rule1 | If collision with the self occurs, then report it and store all the colliding objects. |
| Rule2 | If traffic congestion occurs on road I-94 towards Minneapolis from St Paul, then report it and store the time. |
| Rule3 | While driving, notify when intersection with University & 280 is within two blocks |
| Rule4 | Notify when fuel < 1/4 tank. |
| Rule5 | Speed > 65 miles for any vehicle on the road where I am driving |

**Table 9**
Function $F12$

$$\text{track} = \text{integer}(t * f1(x)) \text{ if } x \text{ inside } [-L_x, L_x]$$
$$\text{track} = \text{integer}(t * f2(x)) \text{ otherwise}$$
$$\text{sector} = \text{integer}(s * f1(y)) \text{ if } y \text{ inside } [-L_x, L_x]$$
$$\text{sector} = \text{integer}(s * f2(y)) \text{ otherwise}$$

$$\text{where } f1(x) = \frac{\mu x}{2L_x} + \frac{1}{2}$$

$$f2(x) = \frac{1}{1 + \exp(-x/\theta)}, \text{and } \theta = \frac{L_x}{\log\frac{1+\mu}{1-\mu}}$$

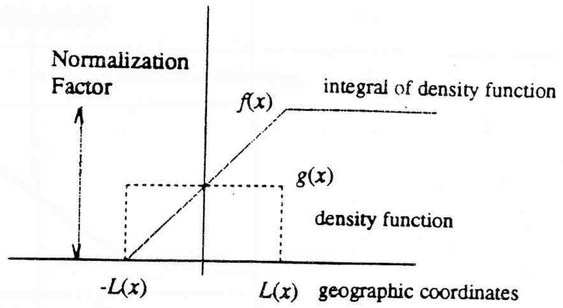

**Fig. 4.** Population distribution and its integral.

integral to the available disk address space. An example normalization factor is the total population of objects. The normalized integral of population density is a monotonic, bounded function providing leotard fit for well-behaved population density functions.

Let us consider the simple object population distribution shown in Fig. 4. The population is uniformly distributed in the interval $[-L_x, L_x]$. There is no population outside the interval. The population density function is represented by $g(x)$ defined by Table 10. The normalized integral is represented by $f(x)$, which is normalized to yield a value between 0 and 1.

The normalized integral can be divided into equal parts for the number of disk blocks available for storing the objects.

The population density functions are defined differently for static and dynamic objects. Population density of static objects (number of objects per unit geographic area) determines the mapping function. The integral of object population density with respect to geographic area gives the mapping function. Let us illustrate it by taking a simple example. Suppose the population density is uniform over the interval $[-L_x, L_x]$ and zero elsewhere. The mapping function will be: disk-address = (number of disk blocks/$(2*L_x)$)$*x$ for $|x| < [-L_x, L_x]$ and zero elsewhere as illustrated in Fig. 4.

Population density functions for moving objects are ill-defined due to the time-dependence of the locations of the moving objects. We use traffic density (i.e., time average population density) to create the mapping functions for the mobile objects. Traffic density function is defined as the time-average of population density of mobile objects in a given area and given time-interval for arbitrarily small area. The traffic density function in general depends on the time interval. However, two time intervals are the most interesting: peak hours, off-peak hours. Our experience with traffic data from the freeways of

**Table 10**
Population integral

$$g(x) = 1 \text{ if } x \text{ in } [-L_x, L_x]$$
0 otherwise.

$$f(x) = \frac{\int_{-\infty}^{x} g(x)\,dx}{\text{Normalization Factor}}$$

$$= \frac{x + L_x}{2 * L_x}, \text{if } x \text{ in } [-L_x, L_x]$$

0 otherwise.

Twin cities confirms this pattern. One may choose a combination of peak-hour and off-peak hour density functions to balance the disk storage costs and computation cost due to overflow costs. For simplicity, we will work with the peak time traffic density function and assume it to be stationary. The integral of the traffic density function with respect to the geographic area gives us the mapping function as well.

A linear combination of the two MoBiLe functions, for static objects and mobile objects, provide the final mapping function used by the database.

## 6.5 MoBiLe Files

MoBiLe file store objects in the geographical world on disk preserving proximity relationships. For 2-dimensional geographic world, MoBiLe files can be designed simply by mapping the two dimensions to tracks and sectors within tracks. For higher dimensions, the mapping becomes non-trivial. In the 2-dimensional MoBiLe File object's geographical location $(x, y)$ is used as the primary key to locate the disk block where the object is stored.

There are four parameters to be determined to design a MoBiLe File: population distribution, mapping function, number of disk blocks needed, and disk allocation. The *population distribution* is derived from population density function. Exact population density for Intelligentan Vehicle Highway System is available from city planning offices. Maps also provide an accurate population density function. The objects occupy two dimensional rangion geographic space. This creates problems in constructing density functions, which specify population values at various geographic points. There could be two ways of resolving the issue: centroid as location, or range function with multiple counting of objects. Centroid as location replaces two dimensional objects by a point object located at the center of gravity of the object. The transformation helps in creation of point density function by specifying the number of objects at each geographic point. An alternative mapping function could be based on creating a set of ranges or a grid. An object will be counted several times, once for each grid overlapping with the two dimensional space occupied by the object. Centroid method is computationally less expensive, but leads to small amount of overflow due to the object located at the boundary of disk blocks.

*Mapping function* from geographic coordinates to disk addresses are based on population distribution. Population distribution function can be exact or approximate. Exact population distribution function can be obtained by numeric integral of exact population density function. An approximate population distribution can be constructed from city maps. The approximation may be based on closed form functions or sampling. A closed form function approximation is based on a set of simple closed form functions whose linear combination can fit population distributions. Sampling uses a collection of points on population distribution curve to represent the function. Interpolation is used to compute the value of population distribution at other points in the populated region. Closed form function approximation leads to a simple mapping function, which has minimal computational overhead during data access. However, it leads to small amount of overflow for the disk blocks mapping the geographic regions with higher approximation error.

*Disk Space Requirement* is determined from the total population, computed from the integral of population density. Other factors determining the disk space requirements are number of bytes needed to store an object, disk block size, overflow requirements and disk utilization factor. Overflow area is needed to account for approximation errors in the mapping function and centroid method of population density estimation. Disk utilization factor is useful for the management of small population density changes due to moving objects to reduce the overflow. The required number of disk blocks for the application ($D$) is determined by $D=[(N\times b)/B]\times[(1+Of)/U]$. The symbols are described in Table 11.

*Space Allocation Strategy:* Disk space can be allocated in one of three ways: as a 2-dimensional rectangle, as a

**Table 11**
Parameters for disk allocation

| | |
|---|---|
| $N$ | Number of objects in the application |
| $B$ | Number of bytes per sector |
| $b$ | Number of bytes per object |
| $Of$ | Overflow factor |
| $U$ | Utilization factor |
| $T$ | Number of tracks allocated |
| $S$ | Number of sectors per track |

1-dimensional chain of tracks (or sectors), or as a chain of blocks randomly linked together. The shape of disk area should be homeomorphic to the population distribution in the geographic space to preserve proximity and continuity relationships. For example, a square area on disk should be used for uniform population distribution over a square geographic region. The number of tracks and sectors/track can be computed by $D=T\times S$. Two-dimensional rectangular disk allocation for general population distribution remains a hard problem.

When one disk is not large enough to store all the objects, the disk address space may need to be partitioned into several disks. Rather than partitioning the population into disks using left-to-right percentile counting, a better way of partitioning is to partition around the peaks of the distribution function and go outward to both directions. A possible benefit of such a method of growing around maxima of population density is that, when focused on one centralized area such as downtown, the queries would not invoke several disk accesses. Typically the number of tracks is larger than the number of sectors/track. This constraint is addressed by mapping the dimension of larger extent to tracks. Furthermore, we logically increase the number of sectors per track by merging two or more physically consecutive tracks into one logical track.

Buffering is used to reduce disk access. A new disk access is needed only when a vehicle crosses over the boundary of blocks. When cross-over occurs, the new block is read into the buffer and the following move operations may work on the buffer rather than actually access the disk blocks.

*MoBiLe File Operations:* Five operations are defined for the MoBiLe Files. They are *build, move, search (collision detection), insert* and *delete*. Each operation is called given a list of parameters, which is put in a parenthesis following the function name. Each object's location is represented as a box with four corners ($c0, c1, c2,$ and $c3$). The upper right corner is $c0$ and the counting of the other corners goes clockwise.

```
Build (a list of object records)
    For each obj in the list, Insert(obj).
```

```
Move (vehicle, new location)
    1. Delete (vehicle).
    2. vehicle-> location = new location.
    1. Insert (vehicle).
Insert (obj)
    1. For each of the four corners, get its block number (b1, b2, b3, b4).
    2. For each block located within the rectangle enclosed by b1, b2, b3, and b4, search the buffer list for the block.
    3. If found, insert obj into the buffer.
    4. Otherwise, copy_obj_from_disk(block_spec), goto 3.
Delete (obj)
    1. For each of the four corners, get its block number (b1, b2, b3, b4).
    2. For each block located within the rectangle enclosed by b1, b2, b3, and b4, search the buffer list for the block.
    3. If found, delete obj from the buffer.
    4. Otherwise, copy_obj_from_disk(block_spec), goto 3.
Collision Detection (vehicle)
    1. For each of the four corners, get is block number (b1, b2, b3, b4).
    2. For each block located within the rectangle enclosed by b1, b2, b3, and b4, search for the buffer list for the block.
    3. If found, for each obj in the buffer, if box_overlap(vehicle->box, obj->box) is true, report obj.
    4. Otherwise, copy_obj_from_disk(obj_block_spec), goto 3.
```

The function copy_obj_from_disk manages buffering. It reads the disk block containing specified object into main memory buffers. It also implements buffer replacement policy and buffer allocation policy.

### 6.6 Experiment design

The experiments were conducted on a Sequent machine under DYNIX(R) V3.0.17.9. The R-tree, SGF and MoBiLe algorithms were implemented from the scratch using C language. The R-tree SplitNode algorithm implemented was the LinearSplit version. Insert, Delete, and Search were fully implemented. The simulation worlds were created using a window-based picture creation tool, FIG, whose output was translated to FIG file via the UNIX utility '$f$2p'. Three worlds were created, containing different number of objects and different population distribution. The worlds are composed of uniformly distributed buildings as shown in Fig. 5.

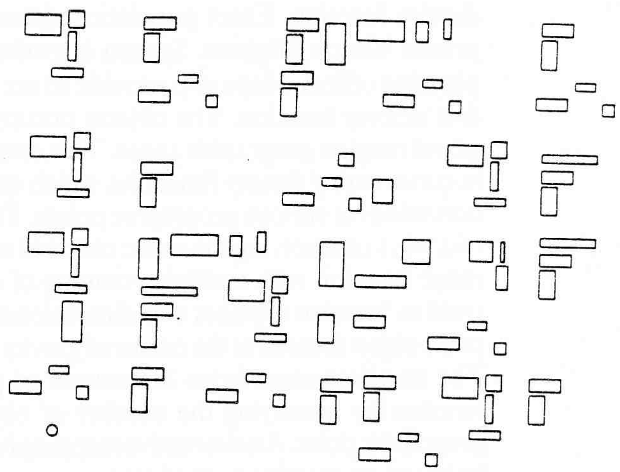
**Fig. 5.** Buildings in centralized world (uniform distribution approximation).

A drive program reads the FIG data file and called the three access methods to create the initial data structure correspondingly and do the 'move object' and 'collision detection' operation according to the instructions read from an operation specification script, which is the same to all trials under different data sets. The times each 'insert', 'delete' and 'search' being called are recorded and updated in an experiment data file. The collected data is then used to compute the disk access cost according to the cost model presented in the previous section.

*Candidates for comparison* are MoBiLe file, R-tree and Simple Grid File. MoBiLe file used mapping function $F12$ to approximate population distribution for centralized population and $F13$ for bimodal distributions to map geographic location to disk addresses. MoBiLe file used one overflow track to account for the error due to approximation of population distribution. We allocated a square-shaped area on disk to the MoBiLe file.

We used R-tree access method and insert, delete, search, and linear split operation as described in Ref. 7. The branching factor of R-tree was determined by disk block size and number of bytes needed to store each object. We experimented with 2 R-trees with branching factors of 32 and 128. The disk accesses were counted for each of the insert, delete and search operations to model the cost of collision detection and motion.

*Simple Grid File (SGF)*[17] was allocated disk blocks, whose actual number is determined by the number of objects in the driving world. A block may contain several cells. The actual disk location of a block is determined by the index of the cell. That is, the geographical location of an object determines which cell it is stored and also indirectly determines which disk block it is stored. For a moving object, its location in the disk needs to be updated every time a movement occurs. The insert and delete operations both take two disk accesses (1 read, 1 write) and search takes one disk access. If cells at the same row are not in the same block, this access time needs to be multiplied by the actual number of block access for the cells; otherwise, only one unit of disk access (2 for insert, delete; 1 for search) is required.

The expected number of block access is determined by the number of cells per block and the cell size (32 bytes in our experiment). Four cases were designed for our experiments: case A, delta = v, block size = 1K, case B, delta = v, block size = 4K, case C, delta = v/2, block size = 1K, case D, delta = v/2, block size = 4K. In case A and B, since the vehicle size is the same as the cell size, for each operation typically 4 cells are touched. For case C and D, 9 cells are touched. In case A and B, the 4 cells are divided into 2 rows. For each row, the probability of both cells being in the same block is 3/4 (1K block contains 4 cells) for case A, and 15/16 (4K block contains 16 cells) for case B. In case C and D, the 9 cells are divided into 3 rows. For each row, the probability of all 3 cells being in the same block is 1/2 and 7/8 respectively. This observation resulted in the following weight functions. The actual time cost function for each operation needs to be adjusted by multiplying the operation count with the following weights. For case A, weight = (1/4)*{4*(1/4) + 2*(3/4)} = 0·625. The weights for other cases are 0·54, 0·5 and 0·37.

*Parameters:* Four parameters for the experiments included disk block size, population size, population distribution, and the number of moving vehicles. Disk block size could take values of 1K or 4K. Chosen values for population size include 0·1K, 0·2K, 1K and 4K. Population distribution was either centralized or bimodal. Centralized population was distributed within a geographically square area. Bimodal population was distributed over two disjoint square-shaped area. Zero or 2 moving vehicles in addition to the self vehicle were used for the experiments. We allocated 32 bytes to store geographic description of each object.

The parameters determined certain aspects of the three access methods. For example, the branching factor of the R-tree was determined by the block size and number of bytes needed to store an object. Object size of 32 bytes yields branching factors of 32 and 128 for block sizes of 1K and 4K respectively. The maximum branching of a node in an R-tree is set to the branching factor. The minimum branching is set to 50% of the branching factor.

*Metric of comparison:* We used disk access cost to compare the performance of alternative access methods. Collision detection on the self vehicle was used as the benchmark task. The query 'Find all the objects colliding with the safety envelope of the self vehicle' can be specified as a sequence of SQL query — *retrieve (object.name) where box_overlap (self.safety_envelope, object.box).* Subsequence queries provide different values for self.safety_envelope to reflect the motion of the self vehicle. We processed the sequence of queries using each access method. The number of disk blocks accessed by the access method is used as the metric of comparison. eN Observations and Analysis.

The number of disk access by the three access methods are presented in Figs 6–11 composed of three groups of bars. Each group has three bars representing costs incurred by R-tree, Grid File and MoBiLe file in processing a common operation. The first group of bars represent the cost of attributed to motion processing for updating locations of moving vehicles. The second group of bars represent the cost attributed to collision detection operation for determining the set of object colliding with self vehicles. The last group of bars represent total cost of processing collision detection in presence of moving

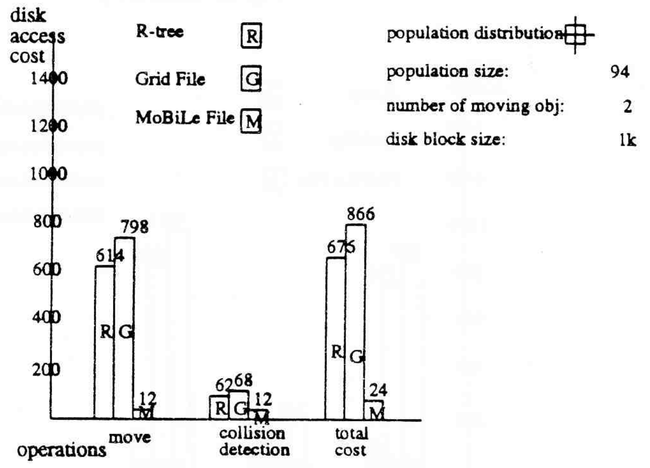

**Fig. 6.** Data set 1.

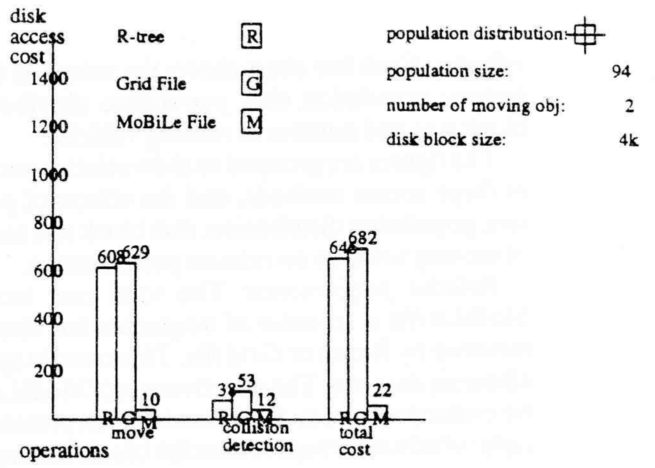

**Fig. 7.** Data set 2.

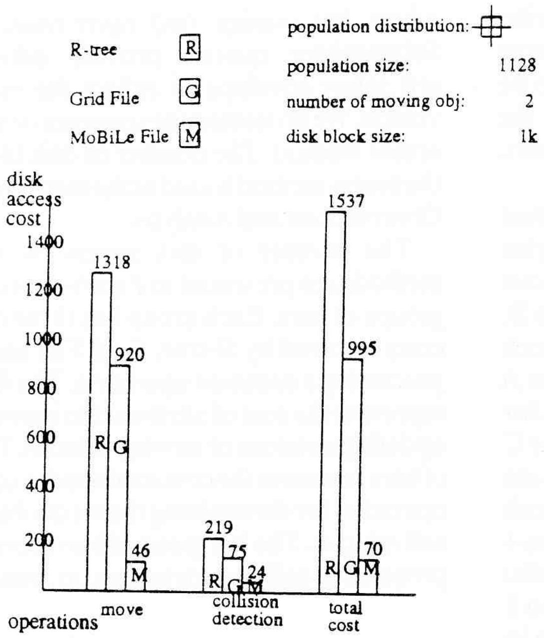

Fig. 8. Data set 3.

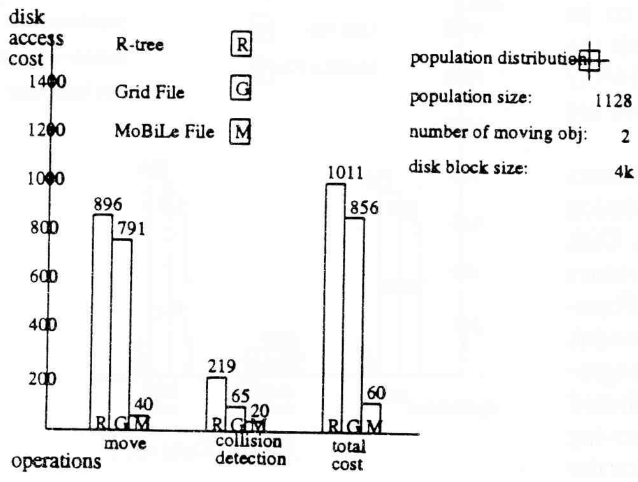

Fig. 9. Data set 4.

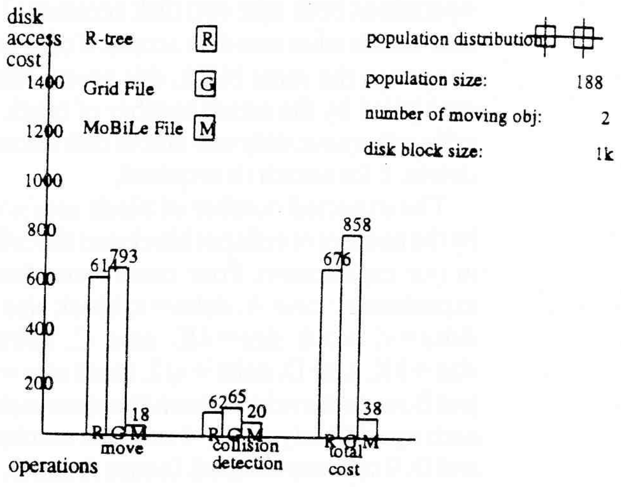

Fig. 10. Data set 5.

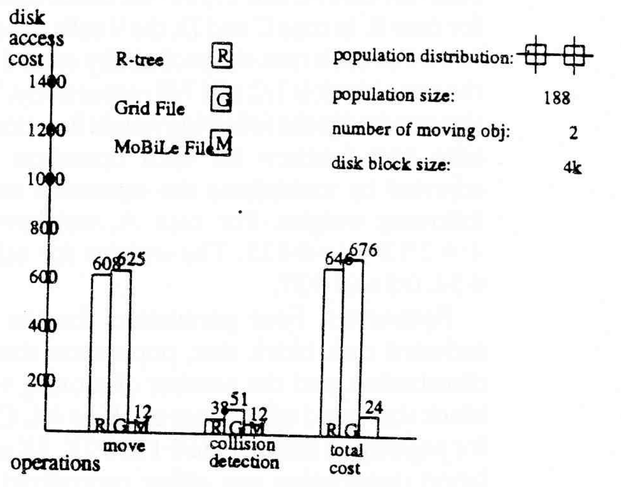

Fig. 11. Data set 6.

vehicles. Each bar chart shows the values of four parameters: population size, population distribution, disk block size and number of moving vehicles.

The figures are grouped to show relative performance of three access methods, and the effects of population size, population distribution, disk block size and number of moving vehicles on relative performance.

*Relative performance:* The total cost incurred by MoBiLe file is an order of magnitude less than the cost incurred by R-tree or Grid file. This trend is apparent in all the six data sets. The effectiveness of MoBiLe files can be understood from the following interpretation. Typically vehicle area maps to one disk block, leading to 1 disk access to answer the query. The cost of entire motion query in MoBiLe access method depends on the number of times vehicle crosses over a disk block boundary. Near the crossover point, vehicle area overlaps with more than 1 block (typically 2) leading to extra disk reads. Every crossover leads to writing the previous disk block (ultimately) and reading the new disk block.

*Effect of moving vehicles:* The effect of moving vehicles on relative performance can be examined from any of the six data sets by comparing the cost attributed to move operation. MoBiLe files access an order of magnitude less number of disk blocks to process motion operations relative to R-tree and Grid file. The cost of processing collision detection query does not show such a marked difference.

*Effect of population size* can be analyzed from data sets 1, 3 and 5 or data sets 2, 4 and 6. Population size does not change the relative performance of the three access methods.

*Effect of population distribution* can be analyzed from comparing data sets 3 and 5 or comparing data sets 4 and 6. Population distribution does not change the relative performance of three access methods. *Effect of disk block size* can be analyzed from data sets 1 and 2, 3 and 4, or 5 and 6. Relative performance does not change with disk block size. MoBiLe file remains the winner, however R-tree improves more than Grid files.

We notice that the time taken (disk accesses) to answer spatial query of collision detection takes constant time in SGF and MoBiLe File. It depends on the branching factor and number of objects in R-tree. The number of disk blocks accessed to move an object is again constant for SGF and MoBiLe File, however it depends on the branching factor and number of objects for R-tree. The performance of SGF and R-tree is comparable. For the case of grid-size being equal to the vehicle-size, SGF does slightly better than R-tree. However for grid-size being equal to half of the vehicle-size, it performs worse than R-tree. MoBiLe file performs much better than both Grid files and R-tree as it uses the information about population distribution.

We have not compared storage and non-disk access computation time explicitly. During our simulation, we noticed that R-tree took very long to execute and SGF leads to very large swap space. MoBiLe Files lead to execution without any of these problems. We estimate that MoBiLe File uses as little space as R-tree and as little compuation time as SGF.

## 7 CONCLUSION AND FUTURE WORK

We have provided a representation of transportation data which is semantically rich in capturing the needs for the driver, traffic controller and researchers. Abstract data types and rules are created using Postgres' type definition and rule definition facilities.

A new spatial access method called MoBiLe Files was created to enhance the geographical data access in a IVHS database application. MoBiLe Files maps from objects' geographical location to the appropriate disk location. Since the mapping function takes into account the population density in the traffic domain, it drastically improves the performance of the database while reducing the disk usage.

We compared the performance of the proposed access method with Grid file, and R-tree to process collision detection query on a moving object. MoBiLe Files seems to perform best in terms of number of disk accesses. In presence of motion, MoBiLe files accessed an order of magnitude less number of disk blocks than traditional methods.

The performance of the database system and the MoBiLe Files access methods needs to be evaluated with real data. Our next step of work is to realize the schema using the Postgres DBMS. The MoBiLe Files access method will be integrated into the system and performance evaluation in a database system will be collected.

**ACKNOWLDGEMENTS**

This research was supported by Center of Transportation Studies and Minnesota Dept. of Transportation under GUIDESTAR project. Our colleagues, namely J. Riedl, J. Carlis, J. Srivastava, M. Shargal and M. Coyle provided excellent comments during the research.

## APPENDIX: RELATION TABLES

**relation name = *vehicle***

| id | name | type | box | safety_envelope |
|---|---|---|---|---|
| car1 | DSY691 | box | bx1 | bx51 |
| truck1 | CND123 | box | bx2 | bx52 |
| self | LV2PLAY | box | bx3 | bx53 |

relation name = *building*

| id | name | business | box | use-hrs |
|---|---|---|---|---|
| b11 | EE/CSci | academic | b10 | 800-1730 |
| b21 | Norris | school | b11 | 900-1800 |
| b13 | Radisson | hotel | b12 | 800-2200 |
| b14 | MinDoT | office | b13 | 900-1700 |
| b15 | UofM Hospital | medical | b14 | 800-2000 |
| b16 | CampusPolice | police | b15 | 24hrs. |
| b17 | Rosedale Mall | shop | b16 | 900-2100 |
| b18 | Fina#126 | gasstation | b17 | 24hrs. |
| b19 | Parking#55 | park'g | b18 | 700-1800 |
| b20 | Metrodome | stadium | b19 | game-schedule |
| b21 | Convention Ctr | stadium | b20 | event-schedule |

relation name = *traffic_sign*

| id | name | type | location |
|---|---|---|---|
| ts1 | signal | point | p21 |
| ts2 | stop1 | point | p22 |
| ts3 | signal2 | point | p23 |

relation name = *traffic_area*

| id | name | type | box |
|---|---|---|---|
| tl1 | rochester | box | bx21 |
| tl2 | eastbank | box | bx22 |
| tl3 | UofM | box | bx23 |
| tl4 | Mississippi | path | bx24 |
| tl5 | Mpls Dntn | box | bx25 |
| t16 | MPLS campus UMN | box | bx26 |

relation name = *traffic_location*

| id | name | point | description |
|---|---|---|---|
| tl1 | Washington&UniversitySE | p31 | intersection |
| t12 | 280&University | p32 | freeway_entrance |

relation name = *sensor*

| d | type | class | location | nebors | occupancyNow | lane | road |
|---|---|---|---|---|---|---|---|
| s1 | point | camera | p1 | s2 | 20 | 1 | r1 |
| s2 | point | camera | p2 | s1,s3 | 30 | 1 | r1 |
| s3 | point | infrared | p3 | s2 | 5 | 1 | r1 |

relation name = *road*

| id | name | path | #lanes |
|---|---|---|---|
| r1 | Washington Av. SE | lseg1 | 2 |
| r2 | Washington Av. S | path2 | 2 |
| r3 | University Av. SE | path3 | 2 |

relation name = *bridge*

| id | name | type | lseg | roadseg | river |
|---|---|---|---|---|---|
| br1 | Washington-Av | lseg | ls10 | r1 | Mississippi |

relation name = *construction*

| id | name | entity | schedule |
|---|---|---|---|
| c1 | repair | r3 | ["Jan 1 12:31:00 1991", "Dec 10 1:00:00 1991"] |
| c2 | create | b19 | ["June 1 8:30:00 1990", "June 1 12:00:00 1991"] |

relation name = *congestion*

| id | type |
|---|---|
| cg1 | event |

relation name = *collision*

| id | type |
|---|---|
| co1 | event |

type name = event

| tid | time | location | participants |
|---|---|---|---|
| e1 | ["May 5 1991", "May 6 1991"] | 11 | Mr. A, Ms. B |

## REFERENCES

1. Blanken, H., Ubema, A., Meek, P. & Van den Akker, B., The generalized Grid File: Description and performance aspects. In *6th International Conference on Data Engineering*, 1990.
2. Codd, E. F., Extending the database relational model to capture more meaning. In *Proc. SIGMOD International Conference on Management of Data*, ACM, 1979, Boston, Mass.
3. Faloutsos, C., Sellis, T. & Roussopoulos, N., Analysis of object oriented spatial access methods. *Proc. SIGMOD International Conference on Management of Data*, ACM, 1987, pp. 426–39.
4. Finkel, R. A. & Bentley, J. L., Quad trees — a data structure for retrieval on composite keys. *Acta Inf.*, **4** (1974) 1–9.
5. Freeston, M., The BANG file: a new kind of grid file. In *Proc. SIGMOD International Conference on Management of Data*, ACM, 1987, pp. 260–9.
6. Greene, D., An implementation and performance analysis of spatial data access methods. In *Proc. 5th International Conference on Data Engineering*, 1989.
7. Guttman, A., R-Trees: A dynamic index structure for spatial searching. In *Proc. SIGMOD International Conference on Management of Data*, ACM, 1984, pp. 47–57.
8. Hinrichs, K. & Nievergelt, J., The Grid File: A data structure designed to support proximity queries on spatial objects, Institüt für Informatik, Eidgenössiche Technische Hochschüle, (ETH), Zurich, July 1983.
9. Hinrichs, K., The grid file system: implementation and case studies for applications, Dissertation No. 7734, Eidgenössiche Technische Hochschule (ETH), Zurich, 1985.
10. Hutflesz, A., Six, H.-W. & Widmayer, P., Globally order preserving multidimensional linear hashing. In *IEEE 4th International Conference on Data Engineering*, 1988, pp. 572–9.
11. Jagadish, H. V., Spatial search with polyhedra. In *6th International Conference on Data Engineering*, 1990.
12. Kriegel, H.-P., Performance comparison of index structures for multikey retrieval. In *Proc. ACM SIGMOD International Conference on Management of Data*, 1984, pp. 186–96.
13. Kriegel, H.-P., Schiwietz, M., Schneider, R. & Seeger, B., Performance comparison of point and spatial access methods. In *Proc. 1st Symposium on Design and Implementation of Large Spatial Database (SSD'90)*, Springer-Verlag, Berlin, 1990.
14. Lomet, D. B. & Salzberg, B., The hB-tree: A robust multiattribute search structure. In *Proc. of the 5th International Conference on Data Engineering*, Los Angeles, CA, Feb. 1989.
15. Lomet, D. B. & Salzberg, B., The hB-Tree: A multiattribute indexing method with good guaranteed performance. *ACM Transactions on Database Systems*, **15** (4) (Dec. 1990).
16. Nagy, G. & Wagle, S., Geographic data processing. *Computing Surveys*, **11** (1) (June 1989) 139–81.
17. Nievergelt, J., Hinteberger, H. & Sevcik, K. D., The Grid File: an adaptable, symmetric multi-key file structure. *ACM Transactions on Database Systems*, **9** (1) (1984) 38–71.
18. Ouksel, M., The interpolation-based grid file. In *Proc. of Symposium on Database Systems, ACM SIGMOD SIGACT*, 1985.
19. Robinson, J. T., The K-D-B-tree: A search structure for large multidimensional dynamic indexes. In *Proc. SIGMOD*, ACM, 1981, pp. 10–18.
20. Roman, G.-C., Formal specification of geographic data processing requirements. *IEEE Transactions on Knowledge and Data Engineering*, **2** (4) (Dec. 1990).
21. Rosenthal, A. *et al.*, Traversal recursion: A practical approach to supporting recursive applications. *Proceedings of SIGMOD International Conference on Management of Data*, (1986).
22. Seeger, B. & Kriegel, H. P., Techniques for design and implentation of efficient spatial access methods. In *Proc. 14th International Conference on Very Large Databases*, 1988, pp. 360–71.
23. Sellis, T., Roussopoulos, N. & Faloutsos, C., The R+-Tree: A dynamic index for multi-dimensional objects. In *Proc. 13th International Conference on Very Large Data Bases*, 1987, pp. 507–18.
24. Shekhar, S. & Yang, T. A., Motion in a geographical database system. In *Proc. 2nd Symposium of Design and Implementation of Large Spatial Database (SSD'91)*, Aug. 1991, Zurich, Switzerland.
25. Silver, J., The GBF/DIME system: Development, design and use, US Bureau of census, Washington, DC, 1977.
26. Stonebraker, M., Inclusion of new types in relational data base systems. In *Proc. Data Engineering*, IEEE, 1986, pp. 262–9.
27. Stonebraker, M. & Rowe, L., The POSTGRES data model. In *Proc. 13th International Conference on Very Large Database*, Sept 1987, Brighton, UK, pp. 83–96.
28. Stonebraker, M., Future trends in data base systems. In *Proc. 1988 IEEE Data Engineering Conference*, Feb. 1988, Los Angeles, CA.
29. Stonebraker, M., *Readings in Database Systems*, Mongan Kaufmann Publishers, San Mateo, CA, 1988.
30. Zobrist, A. L. & Nagy, G., Pictorial information processing of landsat data for geographical analysis. *IEEE Computer Magazine*, **14** (11) (Nov. 1981) 34–41.